LA-UR-03-1560

Approved for public release;
distribution is unlimited.

Title:	ParaView: An End-User Tool for Large Data Visualization
Author(s):	James Ahrens, Berk Geveci, Charles Law
Submitted to:	Technical Report

Los Alamos
NATIONAL LABORATORY

ParaView: An End-User Tool for Large Data Visualization

James Ahrens
Los Alamos National Laboratory

Berk Geveci, Charles Law
Kitware, Inc.

Abstract

This paper describes the design and features of a visualization tool, called ParaView[i], a tool for scientists to visualize and analysis extremely large data sets. The tool provides a graphical user interface for the creation and dynamic execution of visualization tasks. ParaView transparently supports the visualization and rendering of large data sets by executing these programs in parallel on shared or distributed memory machines. ParaView supports hardware-accelerated parallel rendering and achieves interactive rendering performance via level-of-detail techniques. The design balances and integrates a number of diverse requirements including the ability to handle large data, ease of use and extensibility by developers. This paper describes the requirements that guided the design, identifies their importance to scientific users, and discusses key design decision and tradeoffs.

Introduction

Sensors and scientific simulations are generating unprecedented volumes of data making visualization with traditional visualization solutions difficult or even impossible. To address the simulation scientists' visualization needs we spoke with simulation scientists and gathered a set of requirements. The high-level requirements that guided the design of ParaView are support for an efficient workflow and support for the visualization and analysis of extremely large datasets. The challenge was to create a design and implementation that met both these complex requirements and balanced conflicts between them.

Workflow requirements

Visualization is one task of many for simulation scientists. Other simulation tasks include: theoretical work, programming, problem setup, analysis and data management. Therefore, the first workflow requirement is tool ease of use. That is, how long it takes to create results and what visualization domain knowledge is required to run the tool will determine whether the tool is used and how often. A coarse approximation of the simulation scientists' visualization workflow includes two modes: an exploratory mode, in which an interactive graphical user interface(GUI)-based tool is used to explore a dataset; and a batch mode, in which a scripting or programming language is used to write and execute a program that creates an animation. The second workflow requirement is support for both modes. This coarse approximation can be refined further identifying how data is input (during the simulation run or after processing of the simulation is complete) and what type of interface is used (GUI, scripting, VR)[1]. Additional workflow requirements include tool portability, accessibility and extensibility. Portability is required because of the diverse collection of resources available to scientists to run their simulations and visualizations. Tool accessibility is the ability to quickly gain access to, setup, possibly modify and run the tool. Open-source projects are more accessible since the package is typically available on the Internet and any necessary tool modifications can be done quickly because the source is

[i] ParaView is an open-source package and can be downloaded at www.paraview.org.

available. We define extensibility as the ability to easily add new functions and graphical interfaces to the tool.

Large data visualization requirements

The ability to handle large data is also a critical requirement. We define large data as data that exceeds the resource limits (i.e. the elements of the storage hierarchy – memory, disk, tape) of a single machine. The first aspect of the large data handing requirement is a functional one; can the data be visualized at all? Techniques such as data streaming (i.e. processing the data incrementally) and parallelism can be used to process large data sets. Workflow requirements, such as portability, mandate that the tool execute on both shared and distributed-memory parallel machines. The second aspect of the large data handling requirements is performance; can the data be processed quickly? Techniques such as multi-resolution representations and parallelism can be used to improve both visualization and rendering performance.

Related Work

There are a number of visualization packages available for use by scientists. Each of these packages meets a subset of the identified requirements. In this section, we will discuss a few of these packages, specifically AVS[2], OpenDX[3], SCIRun[4] and Ensight[5] identifying their strengths and describing which requirements they meet. ParaView was designed to meet all the identified workflow and large data visualization requirements.

Workflow requirements

Ensight and ParaView use a graphical user interface to execute visualization tasks. AVS, OpenDX and SCIRun use dataflow program graph editor to compose programs. Dataflow program graph editors were thought to provide a good tradeoff between the needs of visualization developers and end-users: for developers, they provide the ability to create complex program graphs and for end-users they provide a graphical interface to create these graphs. In practice, learning visual programming with dataflow graphs is considered by many scientists a significant barrier to creating visualization tasks and thus they consider GUI-based interfaces easier to use. OpenDX, SCIRun and ParaView are all open-source packages making them easily accessible and extensible. These packages offer interactive and batch interaction modes. SCIRun provides support for computational steering – the ability to interact with and visualize data from a running simulation. In contrast to these other packages, ParaView uses a general purpose scripting language, Tcl, for batch commands. Advantages of using a general purpose scripting language include the availability of general purpose computing functionality, robust documentation and support for the scripting language that is independent of the visualization tool.

Large data visualization requirements

All of these packages are portable to most architectures when run on a single machine. Differences arise on their portability to parallel architectures. AVS, OpenDX and SCIRun all support parallel execution on shared memory machines. They also all rely on a centralized executive to allocate memory and execute programs. This reliance makes it difficult to port these packages to distributed memory machines. Ensight uses a client/server architecture, the client renders geometry and the server executes visualization and analysis tasks. Ensight currently provides a shared-memory implementations of both the client and server. Ensight also has a distributed-memory implementation of the server. ParaView is portable to both shared and distributed-memory machines. ParaView is the only listed package that can incrementally process data.

The ability to process datasets larger than the available computing resources is a key consideration when processing extremely large datasets since resource availability changes over time[ii].

Design

ParaView is designed as a layered architecture. The foundation is the visualization toolkit (VTK).[6,7] It provides the foundation of ParaView: data representations, algorithms and a mechanism to connect these representations and algorithms together to form a working program. The second layer is the parallel extensions to the visualization toolkit. The parallel VTK layer, extended VTK to support streaming of all data types and parallel execution on shared and distributed memory machines.[iii] The third layer is ParaView itself. ParaView provides a graphical user interface and transparently supports the visualization and rendering of large data sets via hardware acceleration, parallelism and level-of-detail techniques. Each layer meets a subset of the requirements and adds additional functionality to the layer below.

Visualization Toolkit

The Visualization Toolkit is the foundation of the ParaView architecture. VTK provides data representations for a variety of grid types including structured (uniform and non-uniform rectilinear grids as well as curvilinear grids), unstructured, polygonal and image data. Examples of these grid types are shown in Figure 1. VTK provides hundreds of visualization and rendering algorithms that process these data types including isosurfacing, cutting/clipping, glyphing and streamlines. VTK also provides algorithms for polygon and volume rendering and a keyboard and mouse-based interaction model. Algorithms are organized into dataflow program graphs and a demand-driven dataflow execution model is used to run these programs. Core functionality in VTK is written in C++. To use the toolkit, VTK offers both a C++ library interface and set of scripting interfaces including Java, Python and Tcl interfaces. The library interface provides the best performance. The scripting interfaces offer the advantage of rapid prototyping of programs. Once a day and continuously (i.e. whenever a developer commits a change) tests are run using an open-source testing framework, called Dart, which improves the toolkit's reliability. The toolkit provides the basis for ParaView's portability, accessibility, full range of features and support for interactive and scripting usage modes. More details on VTK can be found in this book in a related chapter on the toolkit.

Parallel and Distributed Visualization Toolkit

Additional functionality was added to VTK to support data streaming and parallel computation.[8] Both depend upon the ability to break a dataset into smaller pieces. Data streaming incrementally processes these smaller pieces one at a time. Thus, a user can process an extremely large dataset with computing resources that cannot store the entire dataset (either in memory or on disk). Data streaming requires that all VTK data types are separable into pieces and the toolkit algorithms correctly process these pieces. To process pieces in a dataflow pipeline, a mapping must be defined that specifies for each algorithm what portion of the input data is required to generate a portion of the output data. With this information, algorithms can generate only a portion of their output for a given input. Each algorithm must ensure program results are invariant, regardless of how the dataset is broken into pieces. These requirements are met by creating a partitioning of both structured and unstructured grid types and by providing ghost levels which are points/cells that are shared between processes and are used by algorithms which require neighborhood

[ii] ParaView's data streaming feature is available in batch mode.

[iii] These extensions are currently part of the toolkit but were added after the original design of the toolkit was complete.

information. A piece of a structured grid is defined by its extent which describes a contiguous range of elements in each dimension (i.e. in 3D, a sub-block of a complete block). VTK's unstructured grid types use an "element of a collection" scheme (i.e. piece M of N). A procedure for converting between grid types was also defined in which each structured extent piece maps one-to-one to an unstructured piece. Additional policies take care of handling boundary conditions and creating ghost levels for all grid types. This data streaming ability supports data parallelism. Instead of processing pieces one of the time, each processor processes a different piece in parallel. Examples of dataset partitioning and the creation of ghost levels are shown in Figure 2. Figure 2 shows a CTH non-uniform rectilinear grid data set that was processed in 8 pieces. The original data set contained cell centered attributes. Volume fraction attributes for both the projectile and plate were first interpolated to vertices before an isosurface filter was used to extract the material surfaces. Both the vertex interpolation and normal generation require ghost cells to ensure partition invariant results.

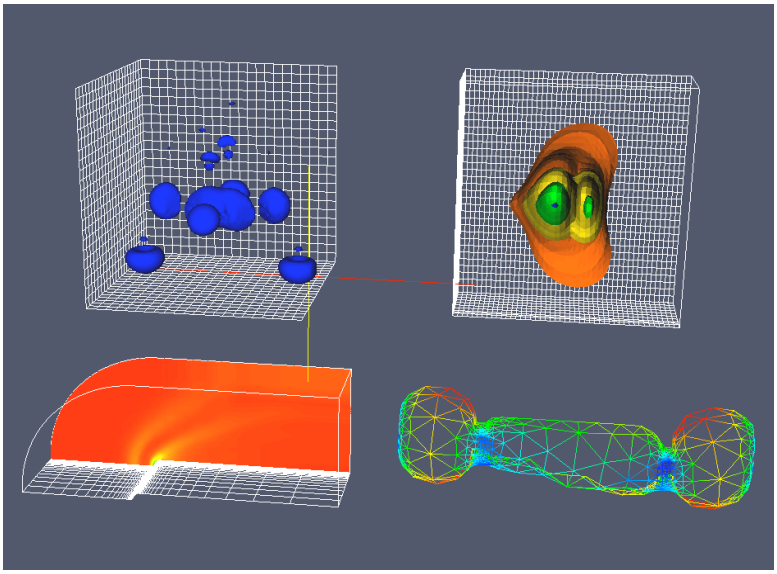

Figure 1: The figure shows the different types of data sets that VTK and ParaView can handle. The upper left dataset is a uniform rectilinear volume of an iron potential function. The Upper right image shows an isosurface of a non-uniform rectilinear structured grid. The lower left image shows a curvilinear structured grid data set of airflow around a blunt fin. The lower right image shows an unstructured grid data set from a blow molding simulation.

Parallel communication and control classes encapsulate details of process initialization and communication libraries such as a shared-memory implementation or MPI. The streaming and parallel computing features can be accessed both thru a C++ library interface and via a set of scripting interfaces. These feature extensions provide the basis for ParaView's large data functionality and performance requirements.

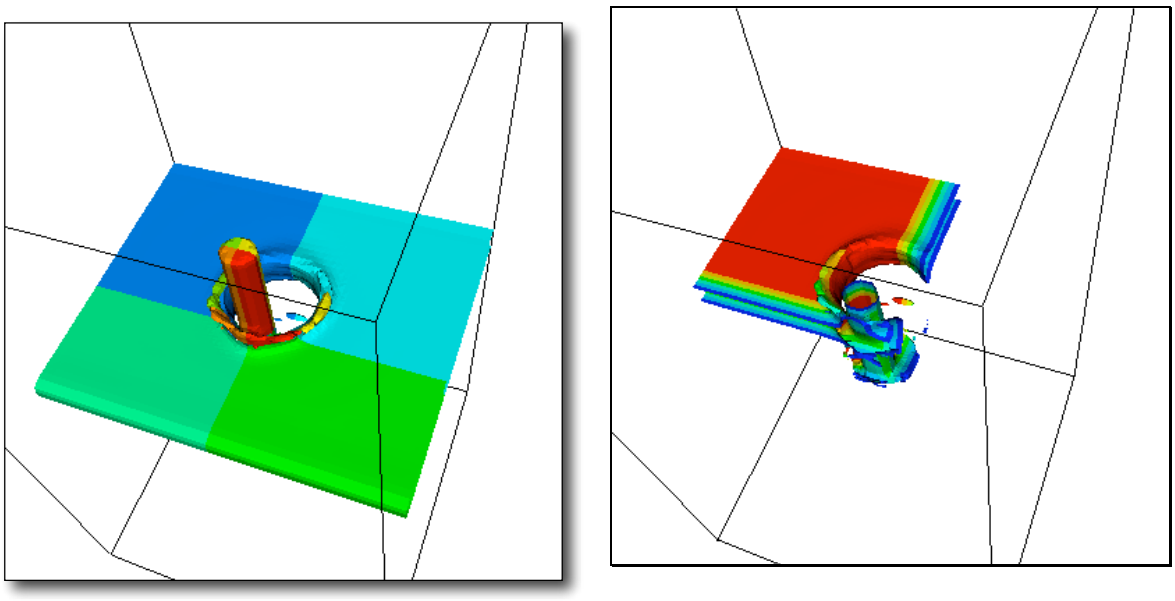

Figure 2: The image on the left was generated with a CTH data set partitioned into eight pieces. Each piece was assigned a different color. The image on the right shows only one partition with six extra ghost levels. The cells are colored by ghost level. In practice, usually only one ghost level is necessary.

ParaView

ParaView provides a graphical user interface for the interactive exploration of large data sets. It builds this functionality on parallel and distributed VTK. An overview of the tool from a user perspective is presented first, followed by a technical description of how the tool's functionality is achieved.

Overview

A sample ParaView session is shown in Figure 3. There are several regions to the user interface including the Menu Bar along the top of the application, the Toolbar just below the Menu Bar, the Left Panel on the left side and the Display Area on the right side. Each of these areas is described in more detail below:

- **The Menu Bar:** The top menu bar provides menu buttons for loading and saving data, creating sources and filters, viewing other windows, displaying help, and other standard functionality.
- **Toolbar:** The Toolbar contains buttons for resetting the camera, switching between 2D and 3D interaction modes, and changing the center of rotation. In addition, the Toolbar contains shortcut icons to instantiate some commonly used filters.
- **Left Panel:** The top portion of this panel contains the selection or navigation window. The selection window provides a list of instantiated sources and filters. The navigation window provides a dataflow program graph representation of the user's task. The area below the selection/navigation window is where the properties of sources and filters are set, which we refer to as a property sheet. Property sheets contain module settings such as the current isosurface values computed by the isosurface module.
- **Display Area:** The Display Area is where the 3D representation of the scene is rendered. Mouse and keyboard interaction are provided in this area.

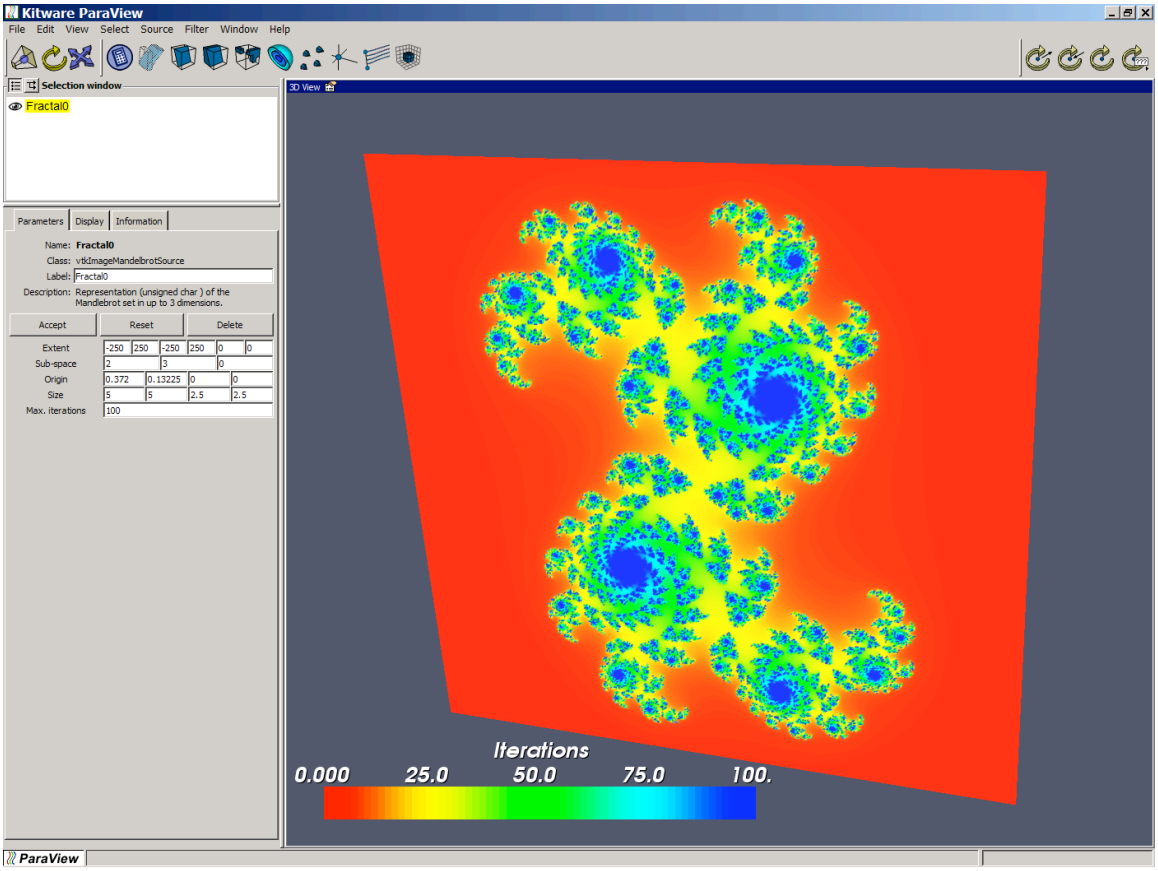

Figure 3: ParaView

To add new filters the user selects a source or filter from the Source or Filter menu on the Menu Bar. Sources include various types of readers or computer-generated sources. A sample of the possible filters includes:

- *Contours and isosurfaces* can be extracted from all data types using scalars or vector components. The results can be colored by any other variable or processed further. When possible, structured data contours/isosurfaces are extracted with fast and efficient algorithms that make use of the structured data layout.
- Vectors fields can be inspected by applying *glyphs* (currently arrows, cones and spheres) to the points in a dataset. The glyphs can be scaled by scalars, vector component or vector magnitude and can be oriented using a vector field.
- A sub-region of a dataset can be extracted by *cutting* or *clipping* with an arbitrary plane, specifying a *threshold* criteria to exclude cells and/or specifying a volume of interest (for structured data types only).
- *Streamlines* can be generated using constant step or adaptive integrators.[iv] The results can be displayed as points, lines, tubes and ribbons and can be processed by a multitude of filters.
- The points in a dataset can be *warped*/displaced with scalars or with vectors.
- With the *array calculator*, new variables can be computed using existing point or cell field arrays. Many scalar and vector operations are supported.

[iv] A parallel implementation of streamlines is not currently available, this feature is under development.

- Data can be *probed* on a point or along a line. The results are displayed either graphically or as text and can be exported for further analysis.

ParaView provides many other data sources and filters by default including edge extraction, surface extraction, reflection, decimation, extrusion and smoothing. Any VTK source or filter can be added to ParaView by providing a simple XML description for its user interface for its property sheet.

The Source and Filter menu are dynamically updated to contain a list of sources/filters that can input the output of the currently selected module. The selected module is either the last module created, or the one most recently selected from the Selection/Navigation Window. Once a module is chosen, a new instantiation of the module is created, connected to the selected module and the module's property sheet is displayed. In this manner, a dataflow program graph is created. In order to manipulate or view the properties of a module, the module is selected and its property sheet is shown and the user can view or edit the listed values.

Meeting the workflow requirements

ParaView simplifies its use by minimizing the knowledge of dataflow programming required by users to use the tool. Specifically, a user can specify simple tasks, for example, creating a source and applying simple filters, without needing to be aware of dataflow programming. This is because ParaView's default behavior is to add new modules to the last module created. When the user wants to change this behavior, for example to apply another filter to the source data, they can use the Selection Window to reset the location where the new module will be added. ParaView also simplifies the choice of modules by only listing modules which accept the correct data type for insertion. For advanced users who wish to create complex program graphs, the program graph is available for manipulation in Navigation window. ParaView is designed so that visualized results dominate the GUI real-estate and the manipulation of program graphs is relegated to a much smaller area. This allows scientists to focus on their visual analysis and not on visual programming which is typically of secondary importance to them.

When modules are instantiated in ParaView they create visual output in the display area providing immediate visual feedback to the user about their data and the effect of the applied module. For example, as shown in Figure 3, when the user creates a two-dimensional source, in this case, a fractal source, ParaView automatically creates a color mapping of the data. This feature improves ease of use because it provides default settings, freeing the user from this task. This feature does have a downside; it hampers the ability to stream data since every module instantiation would cause ParaView to stream visual results. For now, we have chosen to only permit data streaming in batch mode. Solutions to this problem include: offering the option of turning on and off interactive data streaming as well as offering the option of turning on and off the immediate feedback feature.

Users can change the parameters of some modules directly by interacting with the 3D view shown in the Display Area using 3D manipulators. For example, the user can manipulate the seed line of a streamtrace filter by clicking on a control point and dragging the line to the new location. There are also 3D manipulators for probing a dataset with a point or line and cutting or clipping a dataset with a sphere or plane. 3D manipulators improve ease of use by allowing users to quickly apply visualization modules to datasets by letting them interactively use the mouse to select location parameters instead of setting the parameters numerically in the user interface. When the manipulator is adjusted interactively in the 3D view, the numerical values of the location parameters are updated in the user interface and the user can then fine tune these values.

ParaView is portable to most architectures. To achieve this, only packages that work across many platforms were used in developing ParaView. For example, to achieve a portable user interface, Tk, was chosen. Tk, is the graphical user interface companion to the Tcl scripting language. The application framework at the core of ParaView is a unique blend of Tcl/Tk and C++. Tk is used as the widget set but C++ objects, which provide higher-level UI components, are created to encapsulate the widgets. Like VTK objects, these C++ UI objects are automatically wrapped in Tcl.

ParaView's user interface can be modified and extended both statically with XML configuration files and dynamically, at runtime, using the Tcl scripting interface. All ParaView modules and their corresponding user interface are initialized by parsing XML based configuration files. These files contain the input/output types of the modules, an icon name to be displayed on the toolbar, a list of widgets to be displayed on the module's parameter page, the corresponding module parameters and in case of reader modules information about the file type. For example, Figure 4 present the XML description listed in ParaView default configuration file and corresponding user interface for isoline/isosurface module.

```
<Module name="Contour" class="VTKPVContour" module_type="Filter" root_name="Contour"
        button_image="PVContourButton" output="VTKPolyData" input="VTKDataSet">
  <Filter class="VTKPVContourFilter"/>

 <InputMenu id="im" label="Input" trace_name="Input" input_name="PVInput"
        input_type="VTKDataSet"/>
  <ArrayMenu id="am" input_name="Input" attribute_type="Scalars"
        label="Scalars" input_menu="im" number_of_components="1"/>
  <ScalarRangeLabel array_menu="am"/>
  <ContourEntry label="Contour Values" trace_name="Contour Values"/>
  <LabeledToggle label="Compute Normals" variable="ComputeNormals"/>
  <LabeledToggle label="Compute Gradients" variable="ComputeGradients"/>
  <LabeledToggle label="Compute Scalars" variable="ComputeScalars"/>
</Module>
```

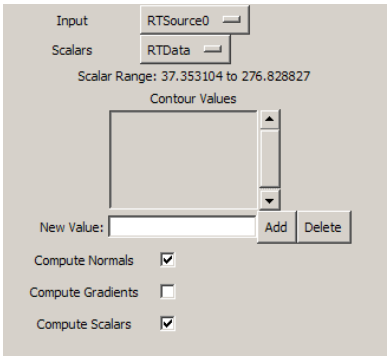

Figure 4: XML description for the isoline/isosurface module in ParaView and corresponding user interface generated by the XML

Since all ParaView widgets have corresponding Tcl representations, the GUI can be modified at runtime by loading scripts or typing commands at a command console. This allows the user to, for example, add new widgets, create dialog windows or load additional libraries at runtime. These features can be used to customize what modules are loaded and how they are presented to the user. A visualization developer can edit the ParaView GUI configuration file and write custom scripts to customize ParaView for use by specific application users. For example, for the climate modeling community, a configuration file could be written to add a suite of climate

analysis modules and customize the existing modules such as contouring to meet community conventions. ParaView meets the accessibility requirement because it is available in open-source form.

Recall that, a coarse approximation of the simulation scientists' workflow includes two modes, an interactive mode, in which an interactive GUI-based tool is used to explore a dataset, and a batch mode, in which a program is executed to create an animation. ParaView supports both these modes. ParaView's interactive mode was detailed in the overview section. Every interaction with the ParaView GUI can be saved in a session file since every interaction has a corresponding Tcl command. The session file can be reloaded to reproduce the session. Furthermore, since the session file is a Tcl script, it can be edited/modified and then reloaded to obtain different results. In addition to dataset exploration, the interactive mode is also used to create to programs to run in batch. ParaView also supports saving the current program graph as a VTK script. A series of queries allow the user to customize the script. A session script differs from a VTK script because a session script saves every ParaView interaction (i.e. every interaction used to create a program graph) and a VTK script saves only a program graph.

Meeting the large data visualization requirements

ParaView supports large data visualization via techniques that include parallel processing, level-of-detail rendering and data streaming.

Parallelism and data streaming

ParaView supports parallelism, either using shared-memory processes or distributed-memory processes via MPI. When ParaView is run in parallel, data is automatically read, processed and, if necessary, rendered in a data parallel manner. ParaView's parallel software architecture includes three types of processes, a client process that runs the graphical user interface and two types of server processes: root and slave processes. The client process communicates with the root server process by broadcasting commands. The root server process receives these commands and re-broadcasts them to all slave servers. A command is a script; currently, a Tcl script is used. In the future, user may be able to select the scripting language to use, for example, Python or java. All servers run an interpreter and use it to execute received commands. This communication mechanism is used to create a copy of the same program graph on each process. The program graphs on the servers manipulate pieces of the full dataset for data parallel processing and the program graph on the client is used to store program state such as parameter settings for modules. ParaView's user interface elements update the client's program graph and the changes are sent as scripts to the root and slave servers. For example, when a user creates a filter module, such as an isosurface, a script is created that instantiates, sets parameters, and appends the isosurface module to a program graph. This script is then communicated to and interpreted by both the client and server processes. All processes use the same naming convention and thus one script works for all process's program graphs. All program graphs are initialized with a rendering module.

To implement parallel algorithms, communication between processes is handled internally by the modules in the program graphs. For example, all rendering modules communicate to implement a parallel rendering algorithm. Although the client can be considered a centralized executive, ParaView's design supports independent process decisions and actions as much as possible. For example, the decision to allocate memory occurs locally. Also inter-process communication is limited to program instantiation/execution commands and parallel algorithms.

ParaView supports data streaming in batch mode. When the user writes a batch script, an option prompts the user as to whether they would like to stream their data and what memory limit they

are bounded by. Streaming along with data parallelism are effective techniques for processing large datasets and effectively fulfill ParaView's large data visualization requirement. We have used these techniques to efficiently and effectively isosurface and color a collection of datasets that ranged in size from 10's of gigabytes to approximately a petabyte in size[8].

Level-of-detail and parallel rendering

ParaView's rendering module supports both level of detail and parallel rendering techniques[9] to facilitate the rendering of very large datasets interactively. Interactive rendering of large datasets is an extremely challenging problem and therefore we applied a number of techniques to achieve good performance.

Level-of-detail techniques increase rendering performance at the expense of image quality. Two different LOD techniques are used in ParaView: geometric LOD and rendered image LOD. The geometric LOD technique creates a reduced geometric resolution model. In general, models with less geometric elements render faster than with more elements. When the user is interacting with model (i.e. rotating, translating, zooming) a reduced resolution model is used in order to render quickly. When the interaction is complete, the full-resolution model is rendered. Figure 5 shows an example of the full and reduced resolution model. VTK's quadric clustering algorithm is used to simplify surfaces. This algorithm preserves data attributes, so the LOD models have the same visual appearance of the original data. Timing results for the decimation algorithm are given in Table 1. Decimation can introduce significant visual artifacts in the rendered image. However, we have concluded that these artifacts are acceptable during interactive rendering. Decimation can also work well with the geometry redistribution technique discussed later in this section. The smaller decimated models can easily be collected and rendered locally.

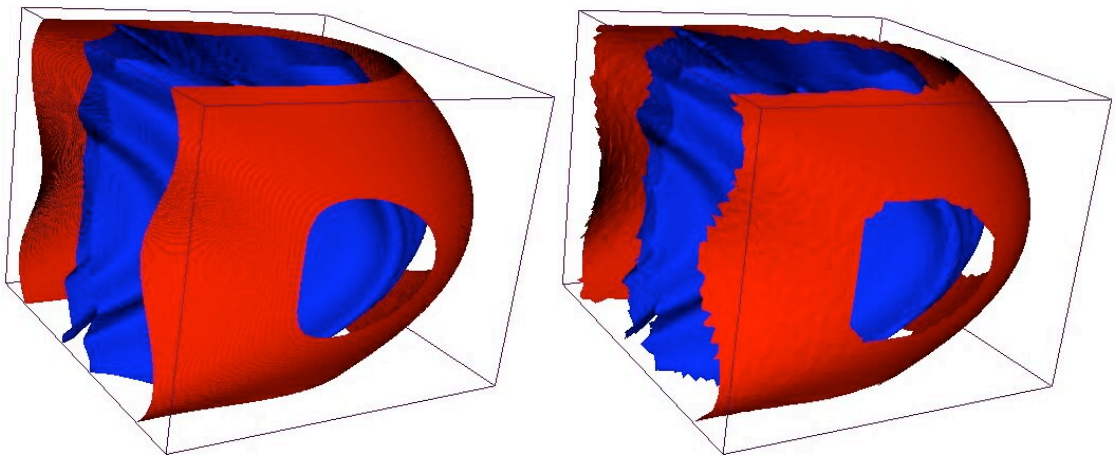

Figure 5: The full-resolution surface (left) has 821,495 triangles and renders in intermediate mode in 0.93 seconds on a GeForce2 GTS, and the decimated surface (right) has 35,400 triangles and renders in 0.04 seconds.

	Time To Decimate
1 Processor	6.25 Seconds
2 Processors	3.515 Seconds
4 Processors	1.813 Seconds

Table 1: The full-resolution surface has 821,495 triangles, and the decimated surface had approximately 35,400 triangles.

The rendered image LOD technique involves rendering to a small image and using pixel replication to magnify the image for final display. It is essentially lossy compression of the

image that has no compression cost and minimal decompression cost. Although the resulting visual artifacts are noticeable, they do not significantly impair interaction with the data set. This technique reduces rendering times through all steps listed below. Initial rendering times can be faster because fill areas are smaller. The time to read and composite image buffers is reduced because it is directly proportional to the area of the rendered image.

Using data parallelism in the renderer is also critical for high performance. The renderer is always in one of two states: an interactive state when the user is interacting with the GUI and a still state when the user is not. The steps of the rendering algorithm are described below. Note that ParaView's geometry and image data can be either serial or parallel data and this may change during execution. The algorithm below is applied to each geometric object to be rendered:

1. **If (Rendering state is interactive) then apply geometric and image LOD algorithm** –
    a. The geometric LOD algorithm is applied when an estimate of the time to render the object (based on the number of points in the object) exceeds a user modifiable threshold. When the threshold is exceeded rendering occurs with a reduced resolution version of the object. If a reduced resolution version does not exist one is created.
    b. The image LOD algorithm is applied when the time to render the last frame exceeds a user modifiable threshold. Using the previous frame time as an estimate a new image size is calculated, in order to reduce rendering time to below the threshold.

2. **If (Geometry data is parallel) then apply parallel geometry load redistribution algorithm –** The result of the LOD algorithm is geometry data. If there is parallel geometry data, this data can be redistributed from its current location on the processes to a subset of the processes. For example, if the geometry is small enough (i.e. after it is reduced by Step 1a) it can be more efficient to collect and render the geometry on a single process. This avoids the cost of parallel image compositing (i.e. Step 4). In this future, this step will also be used to balance geometric load across processes for more efficient performance.

3. **Rendering -** The result of the redistribution algorithm is geometry data. A rendering operation then renders this geometry to create an image and depth buffer result of the image size set in Step 1b. Rendering can be serial or parallel, hardware or software-based, and occur either onscreen or offscreen.

4. **If (Image data is parallel) than apply parallel image compositing -** If there is parallel imagery, then this image data is composited together using the depth buffer to resolve conflicts to create a final image. ParaView currently supports a binary tree compositing with an option of using run-length encoding to losslessly compress images for speed[10]. With large window sizes and many processes, this communication time can be the major factor limiting rendering performance. Compositing transmission time grows linearly with render window area and scales logarithmically with the number of processes. This is why both Step 1a and 1b offers methods (collecting geometry to a single process and compositing using smaller images) to either skip or speedup this compositing step.

Notice that different paths through these steps are possible. For example, a reduced level-of-detail model can be rendered locally when the renderer is in the interactive state and the full resolution version of the same model can be rendered in a parallel when the renderer is in the still state. Having the ability to render at different resolutions and speeds, allows the user to interactively focus on an area of interest and then study the details of a full-resolution image and meets the large data rendering requirement.

Results

This section presents visualization results generated by ParaView for several application areas.

Figure 6 shows isosurfaces of the visible woman data set. The 512x512x1734 900 MB dataset is composed of seven sections. Each section is a uniform rectilinear grid generated by a CT scan. Two isosurfaces were extracted, one for bone and one for skin. The skin isosurface was clipped in order to reveal the bone isosurface. On block of the skin was colored by process id to show the data partitioning. ParaView was run with 4 server processes in this example. This example also demonstrates ParaView's ability to process multi-block data sets. Many structured data sets divide the domain into blocks. Each block is configured to get the best resolution sampling for its domain. Since some data sets can have hundreds of blocks, it is important to group these blocks into a single entity that can be filtered in one step. ParaView has group and ungroup filters, which simplify processing of multi-block datasets.

Figure 7 shows streamlines generated by ParaView using a dataset of airflow around a delta wing. This example also shows the 3D line widget used to seed the streamline algorithm. The streamline filter propagates integration across partition boundaries, and can execute in parallel. The delta wing and the contour surface were obtained by extracting sub-grids from the original curvilinear dataset. Since the actual dataset contains only half the wing due to symmetry, a reflection filter was applied to all surfaces. Both surfaces were colored by mapping the stagnation energy through a default lookup table.

Figure 8 shows the results of a batch script on the results from the Parallel Ocean Program (POP). The 3600x2400x40 structured grid model the earth's oceans at 1/10 of a degree resolution. Isosurfaces and extracted geometry are shown and are used to represent land masses. Also a clip plane colored by temperature at a depth of 1140 meters is shown. It is worth noting that climate-specific visualization tools are unable to process datasets of this magnitude.

Conclusions

This paper presents the design of ParaView, an end-user tool for large data visualization. ParaView provides a graphic user interface for visualizing large datasets using techniques that include data parallelism, level-of-detail and streaming to meet its workflow and large data visualization requirements. In the future, there are number of directions to extend ParaView including the incorporation of data streaming into user interface and rendering support of extremely large datasets for tiled display walls. ParaView is an open source tool and members of visualization community are invited to add new features.

Acknowledgments

This work was supported by grants from the US Department of Energy ASCI VIEWS program. We acknowledge the Advanced Computing Laboratory of the Los Alamos National Laboratory, where we performed portions of this work on its computing resources.

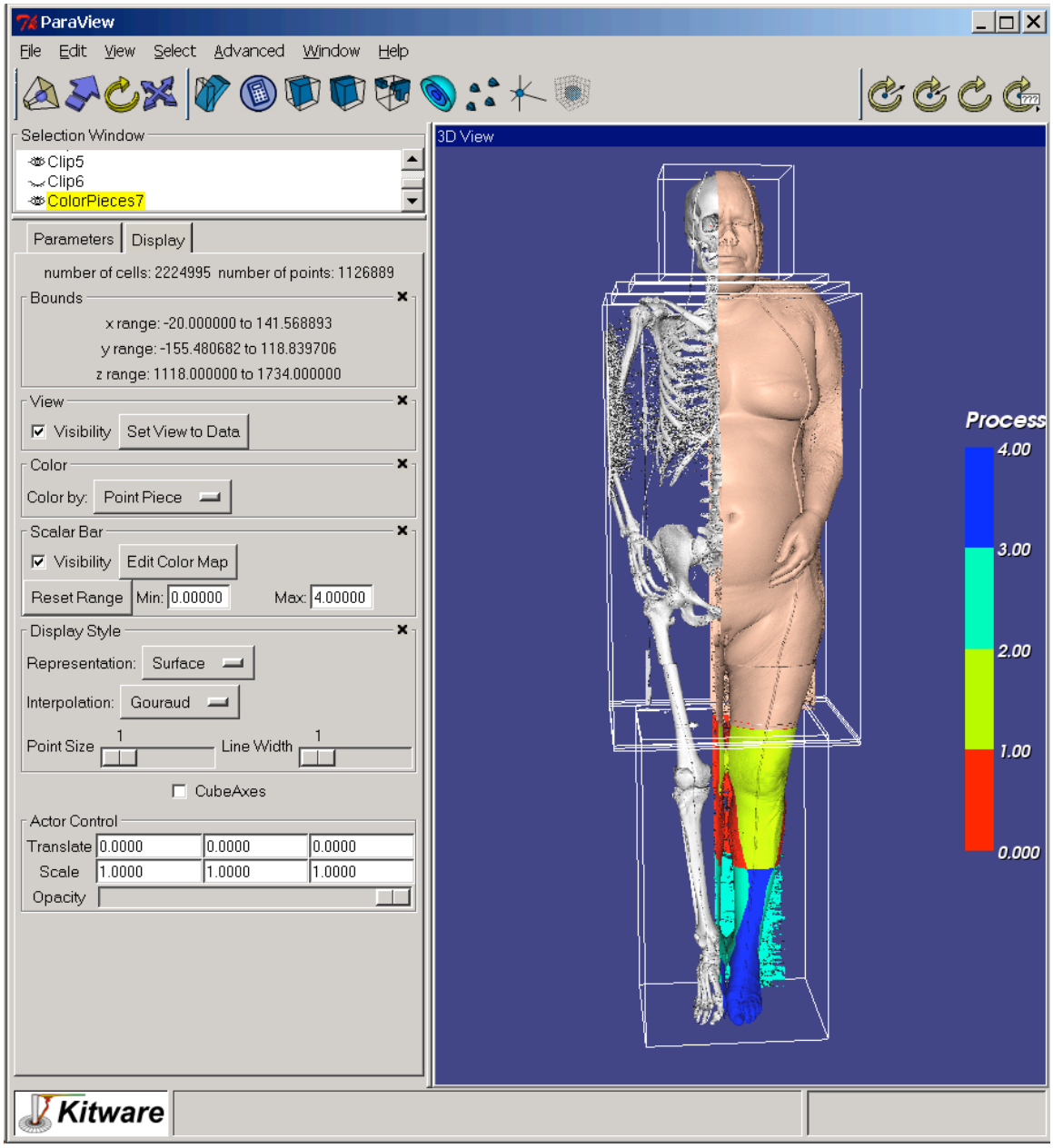

Figure 6: The Visible Woman dataset in ParaView

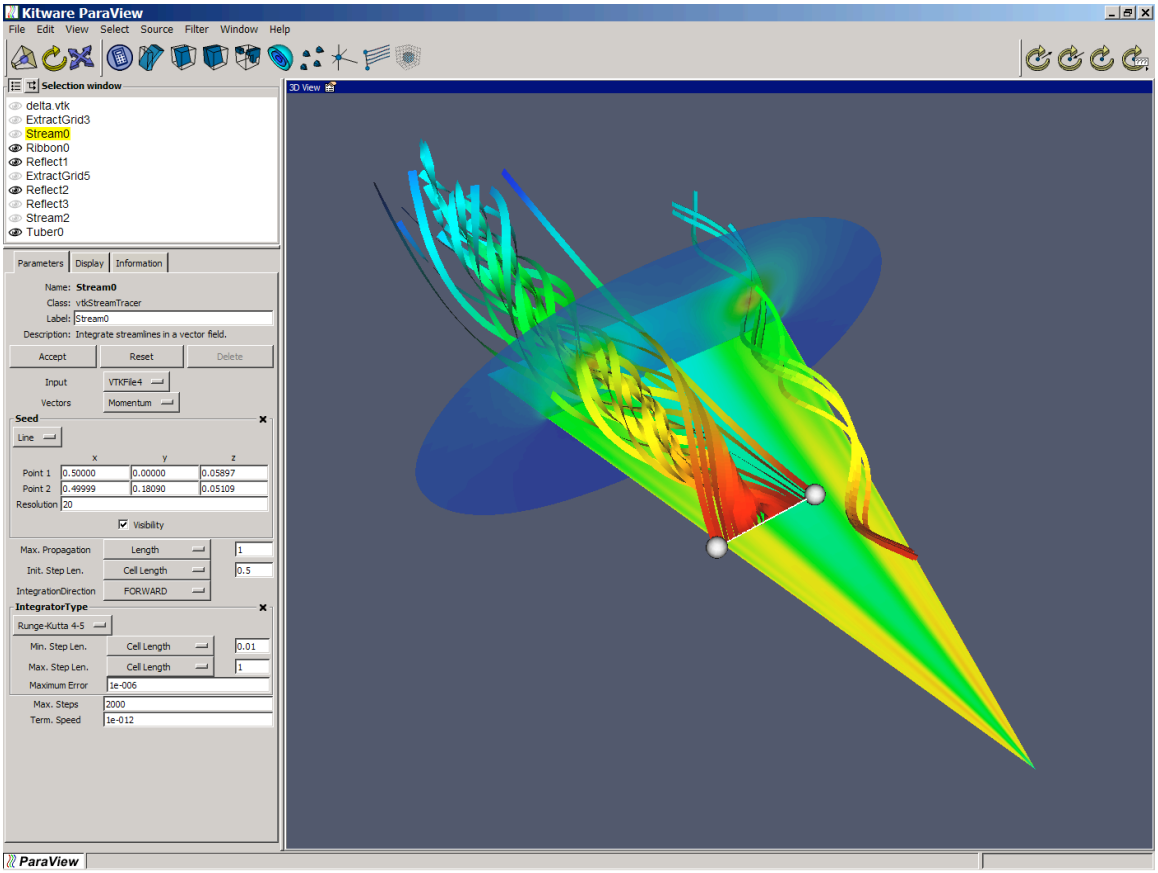

Figure 7: The Delta Wing dataset in ParaView

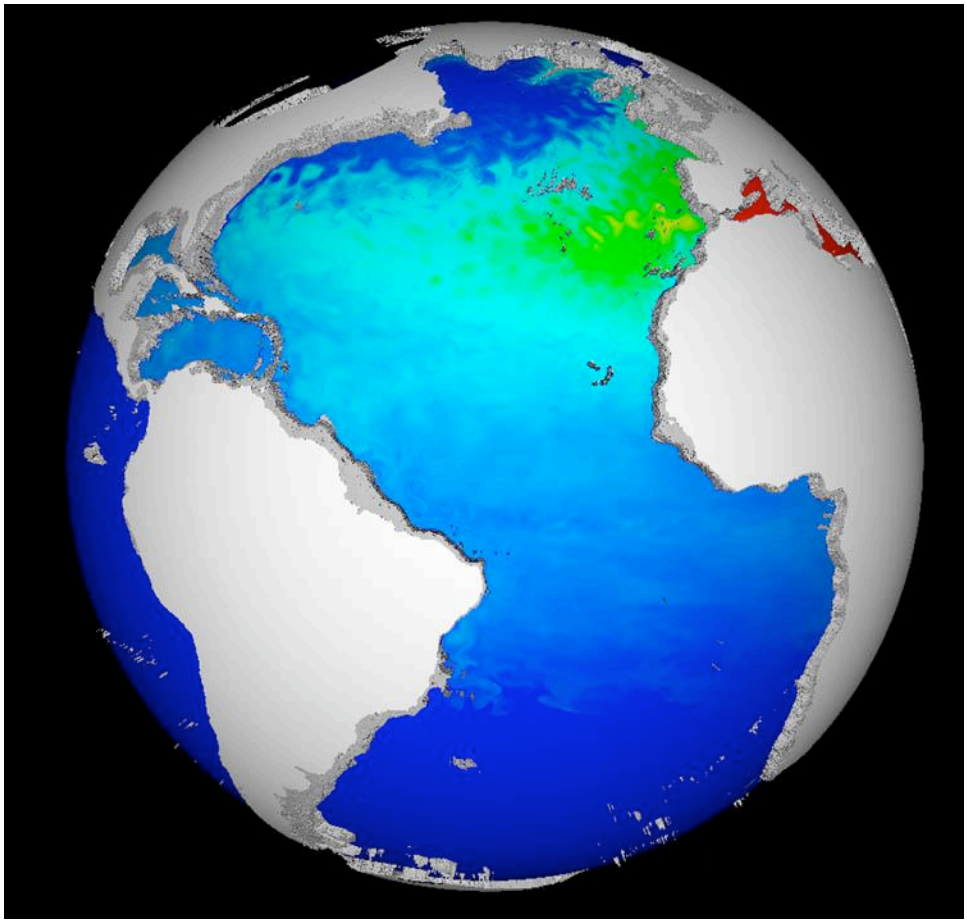

Figure 8: View of the Atlantic Ocean from a global 1/10th of a degree simulation showing ocean temperature at a depth of 1,140 meters generated by a ParaView batch script

---

References

[1] R. Knight, J. Ahrens, P. McCormick, "Improving the Scientific Visualization Process with Multiple Usage Modalities", LAUR-001619.

[2] C. Upson et al., "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics and Applications*, vol. 9, no. 4, July 1989, pp. 30-42.

[3] G. Abrams and L. Trenish, "An Extended Dataflow Architecture for Data Analysis and Visualization," *Proc. IEEE Visualization 1995*, IEEE CS Press, Los Alamitos, Calif.,1995, pp. 263-270.

[4] S.G. Parker, D.M. Weinstein, and C.R. Johnson, "The SCIRun Computational Steering Software System," *Modern Software Tools in Scientific Computing*, E. Arge, A.M. Brauset, and H.P. Langtangen, eds., Birkhauser Boston, Cambridge, Mass., 1997, pp. 1-40.

[5] K. Misegades "EnSight's Parallel Processing Changes the Performance Equation", http://www.ceintl.com/products/papers.html.

[6] W. Schroeder, K. Martin and W. Lorensen, "The Design and Implementation of an Object-Oriented Toolkit For 3D Graphics and Visualization" *Proc. IEEE Visualization 1996*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 263-270.

[7] W.J. Schroeder, K.M. Martin, and W.E. Lorensen, *The Visualization Toolkit An Object-Oriented Approach to 3D Graphics*, Prentice Hall, Upper Saddle River, N.J., 1996.
[8] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. C. Law, M. Papka, " Large-Scale Data Visualization Using Parallel Data Streaming," *IEEE Computer Graphics and Applications*, vol. 21, no. 4, July/August 2001, pp. 34-41.
[9] S. Molnar et al., "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, vol. 4, no. 4, July 1994, pp. 23-31.
[10] J. Ahrens and J. Painter, "Efficient Sort-Last Rendering Using Compression-Based Image Compositing", *Proc. of the Second Eurographics Workshop on Parallel Graphics and Visualization*, 1998, 145-151.